Distributed Multithreaded Caching *D* Compiler

Robert Schadek

CARL VON OSSIETZKY universität OLDENBURG

System software & verteilte Systeme

May 1, 2013

Agenda

1. Introduction
2. Compiler Modifications
    - Overview of Compiler Phases
    - Lexer Parser Communication
    - Multi Threaded Semantic Analysis
    - Caching
    - Distribution
3. The Lexer Generator `dex`
4. The Parser Generator `dalr`
5. Conclusion

Introduction

- basic compiler structure has not really changed since Grace Hopper
- hardware capabilities have improved enormously

Introduction

- basic compiler structure has not really changed since Grace Hopper
- hardware capabilities have improved enormously

- adapt compiler to changed hardware
- learn everything that might be of interest from container to printf style formatting
- graduate

Ideas

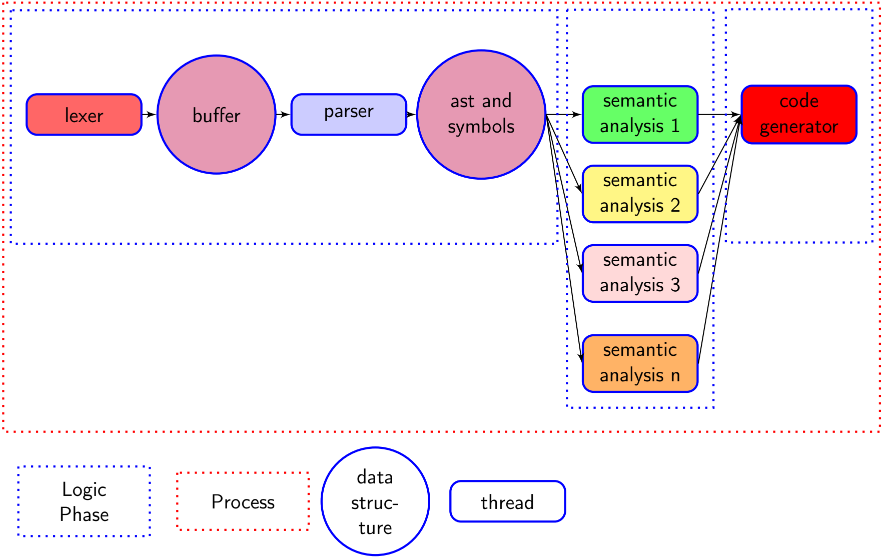
- multithreading: use all CPUs
- caching: use the available RAM
- distributing: distribute work in a network

Ideas

- multithreading: use all CPUs
- caching: use the available RAM
- distributing: distribute work in a network

- lexer generator
- parser generator
- library with container etc.

Overview of Compiler Phases



Lexer Parser Communication

- classic Producer Consumer Problem
- historically a parser asks a lexer for a token

Lexer Parser Communication

- classic Producer Consumer Problem
- historically a parser asks a lexer for a token

- using IO devices interruptively
  - wastes IO performance
  - OS might move HDD head away

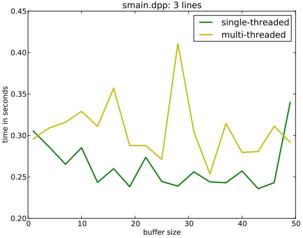
Lexer Parser Communication

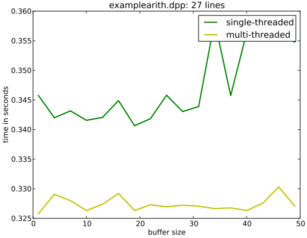
- classic Producer Consumer Problem
- historically a parser asks a lexer for a token

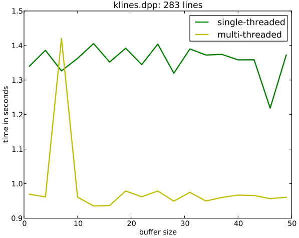
- using IO devices interruptively
  - wastes IO performance
  - OS might move HDD head away

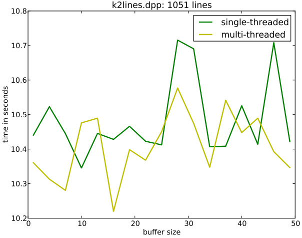
- lexer creates token in a separate thread
- synchronisation is limited by copying multiple tokens at a time

Lexer Parser Communication









Multi Threaded Semantic Analysis

- semantic analysis checks if the program follow the rules
- this is done by traversing the Abstract Syntax Tree (AST) and looking into the symbol table

Multi Threaded Semantic Analysis

- semantic analysis checks if the program follow the rules
- this is done by traversing the Abstract Syntax Tree (AST) and looking into the symbol table

- analysis should not modify any data
- tests are independent of each other

Multi Threaded Semantic Analysis

- semantic analysis checks if the program follow the rules
- this is done by traversing the Abstract Syntax Tree (AST) and looking into the symbol table

- analysis should not modify any data
- tests are independent of each other

- write test as independent functions
- run tests in parallel

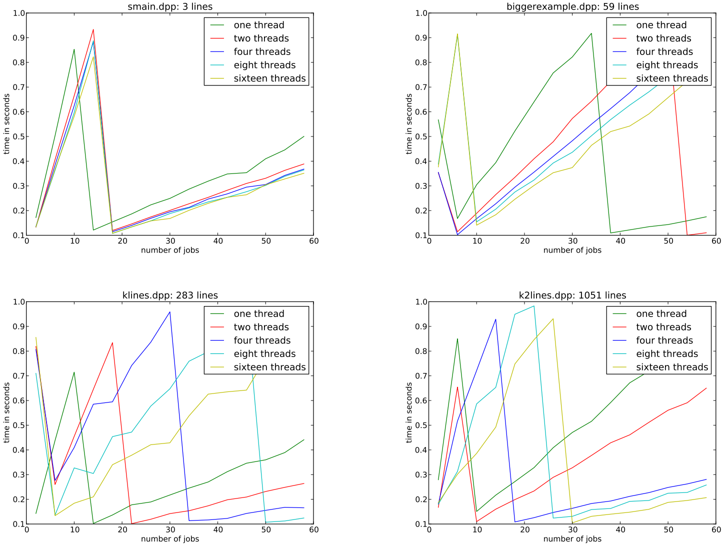
Multi Threaded Semantic Analysis

- semantic analysis checks if the program follow the rules
- this is done by traversing the Abstract Syntax Tree (AST) and looking into the symbol table

- analysis should not modify any data
- tests are independent of each other

- write test as independent functions
- run tests in parallel (without any locking)

Multi Threaded Semantic Analysis Benchmark



Caching

- source files are not independent of each other
- many files get imported many times (e.g. stdio)
- unchanged files do not need to be read from the disk again
- use cached data for distributing work

Caching

- source files are not independent of each other
- many files get imported many times (e.g. stdio)
- unchanged files do not need to be read from the disk again
- use cached data for distributing work

- file level
- token level
- AST level

Caching

- source files are not independent of each other
- many files get imported many times (e.g. stdio)
- unchanged files do not need to be read from the disk again
- use cached data for distributing work

- file level
- token level
- AST level (here it gets interesting)

Linear Trees

Linear Trees

Linear Trees

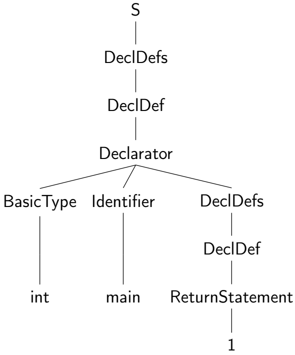
- simplify storing of ASTs in cache
- simplify serializing ASTs

Linear Trees

- simplify storing of ASTs in cache
- simplify serializing ASTs

- flattening complete uniform trees is easy (binary heap, d-ary heap)
- ASTs are neither complete nor uniform

Linear Trees



```
int main() {
    return 1;
}
```

Index	Number of Children	Index of first Child	Name
0	1	0	S
1	1	1	DeclDefs
2	1	2	DeclDef
3	3	3	Declarator
4	1	6	BasicType
5	0	0	int
6	1	7	Identifer
7	0	0	main
8	1	8	DeclDefs
9	1	9	DeclDef
10	1	10	ReturnStatement
11	0	0	1

Array of tree nodes

Index	Child Index
0	1
1	2
2	3
3	4
4	6
5	8
6	5
7	7
8	9
9	10
10	11

Children index array

Linear Tree Benchmark

# nodes	Class based	Struct based
2^{8}	0.0	0.0
2^{9}	0.0	0.0
2^{10}	0.0	0.0
2^{11}	0.0	0.0
2^{12}	1.0	1.2
2^{13}	1.7	4.8
2^{14}	6.1	11.5
2^{15}	10.7	23.0
2^{16}	27.3	47.9
2^{17}	53.4	100.2
2^{18}	109.3	192.7
2^{19}	278.9	403.6
2^{20}	1246.6	815.3

Tree building time in msecs.

# nodes	Class based	Struct based
2^{8}	0.0	0.0
2^{9}	0.0	0.0
2^{10}	0.0	0.0
2^{11}	0.0	0.0
2^{12}	0.0	0.0
2^{13}	0.0	0.0
2^{14}	0.0	1.0
2^{15}	0.0	2.0
2^{16}	2.1	7.3
2^{17}	6.2	19.1
2^{18}	14.3	38.5
2^{19}	31.3	76.7
2^{20}	65.0	154.2

Tree traversal time in msecs.

Distribution

- while developing, many CPUs are idle
- multiplied by the number of workstations in an department
- networks are fast wrt. the size of a source file

Distribution

- while developing, many CPUs are idle
- multiplied by the number of workstations in an department
- networks are fast wrt. the size of a source file

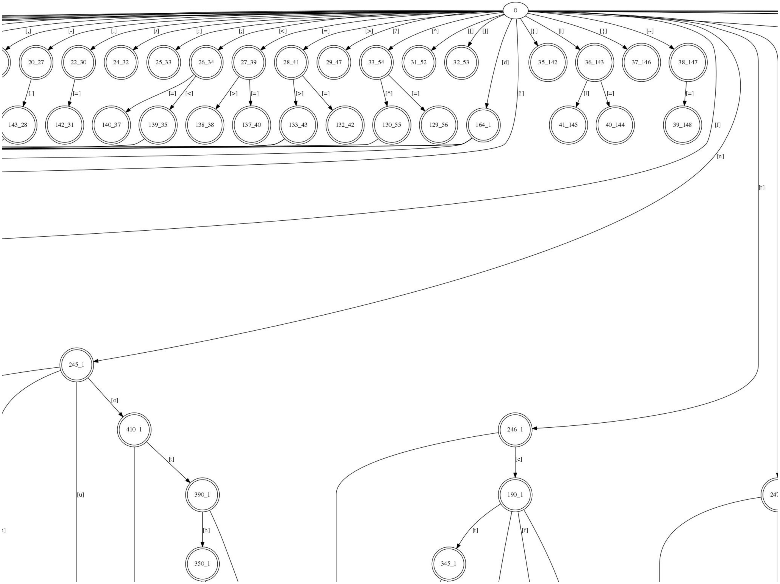
- distribute the compilation of source files to workstations in the network
- compiler becomes a daemon

The Lexer Generator dex

The Lexer Generator dex

- deterministic finite automaton (DFA) tokenizer
- table driven
- user can supply error recovery function
- supports UTF-8
    - transition table is compressed

The Lexer Generator dex



The Lexer Generator dex

state mapping

state	row
0	0
4	1
5	2
7	3

input mapping

input	column
a	0
b	1
c	2
d	3

transition table

	0	1	2	3
0	0	0	4	-1
1	-1	-1	-1	5
2	-1	-1	-1	5
3	-1	-1	7	-1

Original DFA Table

state mapping

state	row
0	0
4	1
5	1
7	2

input mapping

input	column
a	0
b	0
c	1
d	2

transition table

	0	1	2
0	0	4	-1
1	-1	-1	5
2	-1	7	-1

Minimized DFA Table

The Parser Generator `dalr`

- (g|la)lr1 parser generator
- table driven
- accepts all of Chomsky 2 (context free grammars)

The Parser Generator `dalr`

- (g|la)lr1 parser generator
- table driven
- accepts all of Chomsky 2 (context free grammars)
  - user code required to remove ambiguities

Summary

- table driven unicode Lexer possible but infeasible
- splitting lexer and parser works well
- caching has great potential
    - especially with linear trees
- multi threaded semantic analysis is a good approach

Summary

- table driven unicode Lexer possible but infeasible
- splitting lexer and parser works well
- caching has great potential
  - especially with linear trees
- multi threaded semantic analysis is a good approach
  - if not for speed, then at least for clean code

Using D

Using D

What is already there:

- fast turnaround time (crash-fix-build-run)
- compact expressive code

Using D

What is already there:

- fast turnaround time (crash-fix-build-run)
- compact expressive code

What will be there:

- good documentation
- : instead of ..
- containers

The End

The most dangerous phrase in the language is, "We've always done it this way."

Rear Admiral Grace Murray Hopper (December 9, 1906 – January 1, 1992)

```
https://github.com/burner/libhurt
https://github.com/burner/dex
https://github.com/burner/dalr
https://github.com/burner/dmcd
```

```
http://www.svs.informatik.uni-oldenburg.de/60865.html
```